Graph Representations for
Higher-Order Logic and Theorem Proving

Aditya Paliwal[*]
Google Research
adipal@google.com

Sarah M. Loos
Google Research
smloos@google.com

Markus N. Rabe
Google Research
mrabe@google.com

Kshitij Bansal
Google Research
kbk@google.com

Christian Szegedy
Google Research
szegedy@google.com

Abstract

This paper presents the first use of graph neural networks (GNNs) for higher-order proof search and demonstrates that GNNs can improve upon state-of-the-art results in this domain. Interactive, higher-order theorem provers allow for the formalization of most mathematical theories and have been shown to pose a significant challenge for deep learning. Higher-order logic is highly expressive and, even though it is well-structured with a clearly defined grammar and semantics, there still remains no well-established method to convert formulas into graph-based representations. In this paper, we consider several graphical representations of higher-order logic and evaluate them against the HOList benchmark for higher-order theorem proving.

1   Introduction

Mathematics poses a particularly attractive learning challenge, as it can be seen as a test-bed for general-purpose reasoning. HOList (Bansal et al. 2019) is a recently published learning environment for mathematics consisting of a stateless theorem proving API, a benchmark consisting of over twenty thousand mathematical theorems and their proofs, and a neural theorem prover called DeepHOL. It builds on HOL Light (Harrison 1996), an interactive theorem prover that has been used to formalize several mathematical theories, including topology, multivariate calculus, real and complex analysis, geometric algebra, measure theory, and the Kepler conjecture (Hales et al. 2017). The HOList environment allows us to measure progress in automated mathematical reasoning, in particular for machine learning techniques.

The percentage of theorems that can be proven automatically by the neural theorem prover DeepHOL presented by Bansal et al. (2019) appears to be comparable with state-of-the-art algorithms in higher-order reasoning, which typically build on advanced backtracking search algorithms and special-purpose proof calculi (Kaliszyk and Urban 2014; Bentkamp et al. 2018). While these results were very promising, the reference models presented by (Bansal et al. 2019) are still relatively naive—in fact we show that we can

beat their best models with a bag-of-words model. The quest for model architectures that show non-trivial understanding of higher-order logic is thus wide-open.

In this work, we explore the use of the tree structure (e.g. the abstract syntax tree) of logic expressions for learning representations. Most earlier works in this area used TreeRNNs with the idea that the embedding of each node needs to summarize the semantics of the subexpression. However, TreeRNNs (and likewise sequence models such as LSTMs) have not shown strong performance gains over baselines on logical reasoning tasks. We believe this is because TreeRNNs fail to consider the *context* of subexpressions—when computing the embedding of an internal node in the syntax tree, TreeRNNs only consider the embeddings of its child-nodes, but never the parent.

In this paper, we consider the syntax trees of formulas as graphs and apply message-passing graph neural networks (GNNs) (Scarselli et al. 2009; Li et al. 2015; Gilmer et al. 2017; Wu et al. 2019) instead of TreeRNNs. We focus on imitation learning, i.e. learning from human proofs. The proofs we learn from in the HOList dataset were written by human mathematicians interacting with a computer-based theorem prover. To ensure that we make meaningful progress, we evaluate all our models by measuring how many theorems they manage to prove when integrated with the DeepHOL neural theorem prover. In Section 5 we find that GNNs significantly improve performance, and achieve state-of-the-art performance for higher-order logic proof search by a wide margin. Our best model automatically proves nearly 50% of theorems in the validation set.

We present and compare several graph representations for higher-order logic (Section 3) and suggest a simple, message-passing, GNN architecture (Section 4.1). We demonstrate that minor changes in graph representations can have significant effects on the performance of message passing GNNs. We additionally confirm our hypothesis that the context of subexpressions is crucial, by first restricting GNNs to pass messages only upwards in the syntax tree (similar to TreeRNNs), and, second, to pass messages only downwards in the syntax tree. The experiment clearly shows that the second approach, which emphasizes the context of expressions, outperforms the first.

---

[*]Google AI Resident

2 Related Work

Graph Neural Network (GNN) is an umbrella term for several kinds of neural networks that operate on graph-structured data (Wu et al. 2019; Gilmer et al. 2017; Xu et al. 2019). This family of neural networks has been successfully applied in several domains ranging from computer vision (Raposo et al. 2017; Santoro et al. 2017), to predicting properties of molecules (Gilmer et al. 2017), to traffic prediction (Li et al. 2017). Battaglia et al. (2018) show that GNNs are capable of manipulating structured knowledge in the data and hence are a natural choice to learn meaningful representations for higher-order logic.

Similar to our work, Wang et al. (2017) apply GNNs to higher-order logic; they use graph representations of higher-order logic formulas for premise selection, resulting in a significant improvement in prediction accuracy. Our work extends their contributions by using GNNs not only for premise selection, but for predicting tactics and tactic arguments at every step of the proof. While the difference seems minor, it has a big effect on our contributions: it allows us to evaluate our models on a theorem proving benchmark where we demonstrate that our models actually prove more theorems. Because Wang et al. (2017) only predict the premises at the theorem level, they can not generate each individual proof step, and are therefore not able to use their models to generate proofs; instead they use a proxy metric that measures against existing human proofs. We are able to provide end-to-end metrics on the percentage of theorems from the validation set that can be proved using our models. This ensures that our models are learning something useful about mathematics instead of learning to exploit syntactic tricks on the data.

Our findings also differ from the results of Wang et al. (2017) in a crucial aspect: we found that representations that share sub-expressions are significantly stronger than representations that use tree representations. Our graph representation allows for more sharing between expressions, as we merge variable nodes, even if they belong to different binders. Also, our representation includes all type information and has a special node for function applications, which allows us to treat variables, abstractions, and quantifiers in a uniform way.

Similar to this work, GamePad (Huang et al. 2018), TacticToe (Gauthier, Kaliszyk, and Urban 2017), Coq-Gym (Yang and Deng 2019), and Proverbot9001 (Sanchez-Stern et al. 2019) use imitation learning on human proofs collected from the tactics-based higher-order provers Coq and HOL4.

We elected to use the HOList benchmark as opposed to the GamePad or CoqGym datasets as it spans a large variety of mathematics. We additionally wanted to ensure that our models performed well when used to guide proof search, which is made simple in HOList.

Traditionally, automated theorem provers were developed for first order logic. However, most of the serious human formalization efforts (Gonthier 2008; Gonthier et al. 2013; Hales et al. 2017) were performed using interactive proof assistants like Coq and HOL Light (Harrison 1996), which are based on higher-order logic (in which quantification over

arbitrary propositions is allowed). However, the automation of higher-order theorem proving, especially at the tactic level, is a more recent research area. Deep learning has made inroads into various types of logic reasoning, for example to guide SAT solvers (Selsam and Bjørner 2019), QBF solvers (Lederman, Rabe, and Seshia 2018), and for inductive reasoning (Dong et al. 2019).

Many authors have used TreeRNN (or TreeLSTM) architectures to encode logical statements based on their abstract syntax tree representations (Evans et al. 2018; Huang et al. 2018; Loos et al. 2017). While this seems like a natural way to encode tree-structured data, this approach greatly restricts the flow of information across the tree, especially between siblings. TreeRNNs enforce the property that a subexpression will always have the same embedding, regardless of the context in which the expression appears. While this has computational advantages, as it allows embeddings of subexpressions to be cached, it may also limit the ability of the network to effectively encode semantic information across (structurally) long distances.

Conversely, message-passing graph neural networks do not have a single, static embedding for each subexpression, but rather evolve embeddings for every node based on both its children and parents. Perhaps more importantly, when the graph representation allows subexpression sharing, these embeddings can draw on context from multiple occurrences. Our experimental results clearly demonstrate the importance of subexpression sharing.

3 Graph Representations of HOL

In this section we describe our graph representations of higher-order logic (HOL) terms. The HOList benchmark provides its data in the form of S-expressions. For example, the term $f(x)$, which applies a variable f to another variable x, is represented as the following string: (a (v (fun A B) f) (v A x))). The S-expression breaks the function application down into fundamental elements, which we explain in the following: The token a indicates a function application, which always has two children; the function to be applied and the argument. The token v indicates that the function to be applied is a variable. This variable is then described as having type fun A B (i.e. a function mapping from some type A to some potentially different type B) and having the name f. Similarly, the expression (v A x) describes the argument of the function application as a variable x with type A.

There are only a small number of these fundamental building blocks of HOList's S-expressions: a for function applications, v for variables, l for lambda expressions (also called abstractions), c for constants, and some additional ones for type constructors such as fun. All other tokens are names of either variables, constants, or types, of which there are around 1200 in the dataset. Even quantifiers have no special treatment and are simply represented as functions, using the definition $(\forall) \triangleq \lambda f.\,(f = \lambda x.\,True)$.

That is, S-expressions essentially represent the abstract syntax tree of higher-order logic expressions. We consider the abstract syntax tree (AST) as a graph with directional
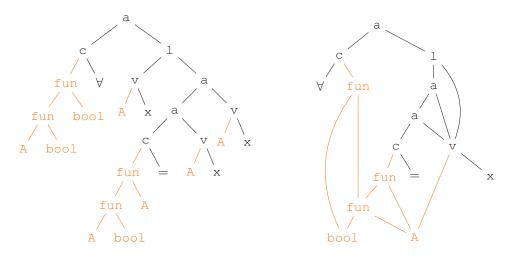
Figure 1: Comparison of two graph representations of $\forall x : \ x = x$. Types (e.g. fun, bool, and type variable A) are printed in orange. Left: AST representation. Right: AST with subexpression sharing (edge labels ensure the left/right order of children is preserved, but are omitted for readability in this figure). Sharing reduces the graph size from 27 nodes to 15; the amount of sharing increases as the terms grow larger.

edges, where each node has edges to its children and also to its parent(s). Edges are labeled with the index of the child, which makes the original string representation recoverable from the graph.

In addition to the plain AST, we consider several modifications to the graph representations, which we study in Section 5:

- We share nodes of syntactically equal subexpressions (*subexpression sharing*).

- We share equal leaves of the AST (*leaf sharing*).

- We replace all variable names by x (*variable blinding*).

- We add random edges (*random*).

- We remove all edges from nodes to their parents and only keep those to their children (*top down*).

- We remove all edges from nodes to their children and only keep those to their parents (*bottom up*).

Subexpression sharing merges all nodes in the AST that represent the same expression. In Figure 1 we illustrate the difference between the abstract syntax tree representation with (right) and without (left) subexpression sharing. The variable x of type A, which occurs three times in the abstract syntax tree, now occurs only once. It is represented by the S-expression $(v\ x\ A)$, and the root node of this expression now has three parents that keep track of the locations of each of its original occurrences. Note that sub-expression sharing also happens over types. In this example, variable x has type A, so every other expression with type or sub-type A is now connected through this node. For detailed statistics of how subexpression sharing compresses the data, see the Appendix of the extended version of this paper (Paliwal et al. 2019).

Leaf sharing is a middle ground between the very verbose AST representation and the aggressive graph reduction that

is subexpression sharing. Instead of merging nodes that represent equal subexpressions, we only merge leaves with the same token, such as A, bool, or x.

Variable blinding replaces all variable names by x. This transformation is applied only after the graph is constructed and does not modify the graph structure. Hence, different variables still have different graph nodes and are thus distinguishable - just their names are the same. This allows us to study how much our networks rely on variable names.

Restricting our graphs to only have *top down* edges or only have *bottom up* edges allows us to study the question of how important the context of subexpressions is for their embeddings. Bottom up loosely resembles TreeRNNs, as it computes its embeddings only from the embeddings of its children. Top down, on the other hand, disallows nodes to see their children, but allows them to see their context.

Adding random edges to the graph representation allows us to study if additional connectivity helps the neural networks to propagate information through the graphs. We chose to add 3 outgoing edges per node to random targets, as this approximates the construction of expander graphs, which provides excellent connectivity properties of the resulting graph with little overhead. We label the random edges to make them distinguishable from regular edges in the graph.

4   Model Architecture

In this section we present our neural network architecture, starting with basics of message-passing GNNs in Section 4.1, which we use for embedding statements in higher-order logic. We then detail the prediction tasks necessary for guiding proof search and our imitation learning approach, which trains on human proofs, in Section 4.2.
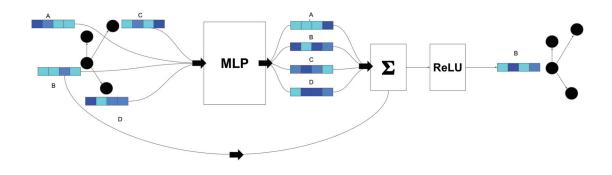
Figure 2: A single Graph Neural Network neighborhood aggregation step for node B. The features over the neighborhood are passed through an MLP, followed by a summation and a non-linearity. The output that follows is the hidden feature for node B for the next step.

4.1 Graph Neural Networks

Graph neural networks (GNNs) compute embeddings for nodes in a graph via consecutive rounds (also called *hops*) of end-to-end differentiable message passing. The input to a GNN is a labeled graph $G = (V, E, l_V, l_E)$ where V is a set of nodes, E is a set of directed edges, l_V maps nodes to a fixed vocabulary of tokens, and l_E maps each edge e to a single scalar indicating if the edge is to (or from) the first or the second child, encoded as 0 and 1. Each token t in the vocabulary is associated with a trainable feature vector $\boldsymbol{x}_t$.

The GNN computes node embeddings $\boldsymbol{h}_v$ for each node $v \in V$ via an iterative message passing process of T rounds:

1. Embed nodes n and edges e into high dimensional space using multi-layer perceptrons:

$$\boldsymbol{h}_v^1 = \mathrm{MLP}_V(\boldsymbol{x}_{l_V(v)})$$

$$\boldsymbol{h}_e = \mathrm{MLP}_E(l_E(e))$$

2. For each round $t \in \{2, \ldots, T\}$, and for each edge $(u, v) = e \in E$, pass the node embeddings from the previous step $\boldsymbol{h}_u^{t-1}$ and $\boldsymbol{h}_v^{t-1}$ and the edge embedding $\boldsymbol{h}_e$ into an MLP to generate messages. When computing the messages for a node v, we distinguish between messages from parent nodes $\boldsymbol{s}_{u,v}$ and from child nodes $\hat{\boldsymbol{s}}_{u,v}$:

$$\boldsymbol{s}_{u,v}^t = \mathrm{MLP}_{\mathrm{edge}}^t([\boldsymbol{h}_u^{t-1}, \boldsymbol{h}_v^{t-1}, \boldsymbol{h}_e])$$

$$\hat{\boldsymbol{s}}_{u,v}^t = \hat{\mathrm{MLP}}_{\mathrm{edge}}^t([\boldsymbol{h}_u^{t-1}, \boldsymbol{h}_v^{t-1}, \boldsymbol{h}_e])$$

3. To summarize the messages for node $v \in V$, we sum over the messages from parents and from children separately before passing them through an MLP and adding them to the previous embedding:

$$\boldsymbol{h}_v^t = \boldsymbol{h}_v^{t-1} + \mathrm{MLP}_{\mathrm{aggr}}\left(\left[\boldsymbol{h}_v^{t-1}, \sum \frac{\boldsymbol{s}_{u,v}^t}{p(v)}, \sum \frac{\hat{\boldsymbol{s}}_{u,v}^t}{c(v)}\right]\right)$$

MLP_V, MLP_E, $\mathrm{MLP}_{\mathrm{edge}}^t$, $\hat{\mathrm{MLP}}_{\mathrm{edge}}^t$ and $\mathrm{MLP}_{\mathrm{aggr}}$ are multi-layer perceptrons, $p(v)$ is the number of parents of node v, $c(v)$ is the number of children of node v $[,]$ is the

vector concatenation operator, and $\boldsymbol{h}_v^t$ represents the embedding of node v after t rounds of message passing. A multi-layer perceptron (MLP), is a function $\mathrm{MLP} : \mathbb{R}^a \to \mathbb{R}^b$, that maps vectors in the input space to the output space via successive applications of linear transformations and non-linear activations.

The final set of node embeddings returned by the Graph Neural Network is given by $\boldsymbol{h}_v = \boldsymbol{h}_v^T$ where T is the number of message passing rounds. These node embeddings represent information from the T-hop neighborhood of each node in the graph. A single step of message passing over a single node of a graph is shown in Figure 2.

4.2 Complete Model Architecture

The full architecture is depicted in Figure 3. The network takes the current goal (an intermediate state in a proof search) and a candidate premise.

It then generates node embeddings $G(g)$ and $P(p)$ for the graph representations of both the goal g and premise p using identical GNNs; GNN-1 and GNN-2 do not share weights. To make the architecture computationally tractable, the depth of each node embedding is relatively small (128). We do two 1x1 convolutions to expand the depth (to 512 and then to 1,024) immediately before max pooling, resulting in a goal embedding and premise embedding, each of size 1,024.

The tactic classifier selects from a fixed set of 41 tactics. It uses two fully connected layers, followed by a linear layer to produce logits for a softmax tactic classifier. The combiner network concatenates the goal and premise embeddings, as well as their element-wise multiplication, followed by three fully connected layers. The model uses sigmoid cross-entropy to score how useful an individual premise is for the given goal. It was important to apply dropout throughout the network.

This architecture was trained on the proofs written by humans and released in the HOList dataset. For evaluation, we plug the models into the breadth-first proof search provided by the HOList environment. We first start with the top level goal which is the top level theorem to be proved. To process a goal (or "subgoals" for newly created goals), we ap-
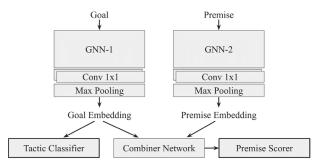
Figure 3: Diagram of the model architecture. At every step of the proof, the current goal is embedded using GNN-1. Based on the goal embedding, the model predicts the next tactic to apply from a fixed set of 41 tactics. The goal embeddings are also used to score every preceding theorem, called premises. Premises are also embedded using a graph neural network (GNN-2). Higher scores indicate the given premise is predicted to be useful for proving the current goal. Full details presented in Section 4.2.

ply one of the tactics that may take a list of already proven theorems as parameters.[1] Once a goal is set to be processed, we pick the top-k_1 highest scoring tactics from the softmax tactic classifier (see Figure 3). Then the combiner network is evaluated for each (g, p_i) pairs of the current goal g and possible tactic parameter p_i (which includes all the definitions and theorems preceding the top level goal in the theorem database). Note that this computation is accelerated significantly by pre-computing the premise-embeddings $P(p_i)$, and therefore only the combiner network has to be evaluated for each pair of embeddings $G(g), P(p_i))$, where $G(g)$ denotes the goal embedding of the currently processed goal. In total, there are 19,262 theorems and definitions in the theorem database. Only the top-k_2 highest scoring premises are chosen as tactic arguments. In our setting, we used $k_1 = 5$, $k_2 = 20$.

A tactic application might fail or may be successful. Failed applications are logged, but ignored for proof search. If the tactic application is successful, it might close (that is: prove) the goal or generate a new list of subgoals (or proof obligations). The original goal is closed if each of its subgoals is closed. During proof search, the HOList proof-search graph maintains several alternative branches of subgoals to be closed. Proof search stops if the top-level goal closes, that is if at least one of the alternative branches closes.

Note that in contrast to earlier premise selection works, we generate new tactic parameter lists for each subgoal during proof-search, not just a single list of premises for the top-level goal.

---

[1]The interpretation of the tactic parameters depends on the type of the tactic. For MESON_TAC, these are premises that the tactic can use for proving the statement in HOL Light's built-in first-order reasoning algorithm. For REWRITE_TAC, these should be equations that are applied for rewriting the goal statement.

Table 1: Our best model, compared against previous state of the art and bag of words model as baselines.

Network Architecture	% Proofs Closed (Validation Set)
Baseline: S-expression as a string WaveNet (Bansal et al. 2019)	32.65%
Baseline: Bag of words Max pooling only	37.98%
Subexpression sharing 12-hop GNN	**49.95%**

5 Experiments

Here we present our experimental results on the HOList benchmark (Bansal et al. 2019), which is based on the complex analysis corpus of HOL Light (Harrison 1996). The training set is derived from the proofs of 10,200 top-level theorems and has around 375,000 proof steps (subgoals) to learn from. We have evaluated the end-to-end prover performance on the standard validation split of HOList which consists of 3,225 held out theorems of the complex analysis corpus.

In total, there are 19,262 theorems and definitions in the HOList corpus. These theorems are proved in a sequential order. So, for any given goal, all preceding theorems and definitions in the corpus are eligible premises. For each tactic application, we select the top 20 premises after ranking the eligible premises with the premise scorer.

5.1 Evaluation Metrics

The primary way that we evaluate our models is by using them to guide the proof search in the HOL Light theorem prover, and then counting how many theorems each trained model can prove. However, it takes several hours to run the prover on the validation set, even in a distributed manner. If we were to use this metric during training, it would slow down training of our models by several orders of magnitude, so we instead train using two proxy metrics:

- We look at the accuracy of the tactic prediction output, which decides which one of the 41 possible tactics is to be applied for a given goal. It is often the case that more than one choice of tactic would work, but to determine if a tactic is correct, we would need to try it in the prover. Instead, for the proxy metric, we only count a tactic correct if it was the exact tactic applied in the human proof, as our data contains only a single proof per theorem.

- We monitor the relative prediction accuracy for the tactic parameter selection, which is the ratio of cases in which a true tactic parameter is scored higher than some randomly sampled parameter. Again, there is often more than one "true parameter", but we only consider the set of premises that were used as tactic parameters in the human proof.

We then perform the computationally expensive evaluation only on the best checkpoint (according to the proxy metrics): running the prover once over the validation set using the parameterized tactics predicted by the checkpoint to guide proof search. This *percentage of proofs closed* in

Table 2: Percentage of proofs closed on the validation set. Representations are defined in Section 3. Two AST results indicated with an asterisk (*) required a smaller batch size due to memory constraints.

Representation	0 Hops	2 Hops	4 Hops	8 Hops	12 Hops
Abstract syntax trees (AST)	40.18%	43.84%	44.58%	46.66%*	45.67%*
Leaf sharing	41.76%	33.89%	29.24%	29.51%	30.51%
Leaf sharing + variable blinding	31.78%	32.18%	32.80%	30.04%	31.00%
Subexpression sharing	40.86%	42.94%	46.94%	47.22%	**49.95%**
Subexpression sharing + variable blinding	31.75%	34.44%	35.96%	34.07%	37.36%
Subexpression sharing + random	41.24%	43.68%	43.84%	42.63%	42.94%
Subexpression sharing + top down	40.55%	43.59%	45.51%	48.24%	48.40%
Subexpression sharing + bottom up	39.72%	40.58%	41.16%	41.86%	40.99%

the validation set is the primary metric that we use to establish the proving power of a given checkpoint. It evaluates whether a trained model is effective at automatically proving newly encountered theorems.

5.2 Results and Analysis

We train several models using the architecture described in Section 4.2, varying the number of message passing rounds in the GNN and the representation of the higher-order logic formulas as graphs as described in Section 3.

We compare our best-performing model against baselines in Table 1. Our 12-hop shared subexpression GNN achieves the state-of-the-art result on the HOList benchmark, closing 49.95% of the proofs. This is a major improvement on the baseline models in the HOList paper (32.65%) and even outperforms the reported 38.9% of their best reinforcement learning experiment. We also observe that a simple bag-of-words model with large word embedding sizes also manages to close 37.98% of all proofs. The GNN models with zero message passing steps (0 Hops) are roughly equivalent to the bag of words max pooling model, except that the 0 Hops models also have additional MLPs between the word embedding lookup step and the max pooling step.

In Table 2 we can see that the number of proofs closed increases with additional hops (message passing steps) for ASTs with and without subexpression sharing. This indicates that additional information about the structure of higher-order logic formulas provides the model with the capacity to learn better embeddings.

Variable blinding, which initializes every token representing a variable name to the same (trained) embedding, allows us to evaluate the importance of human-assigned variable names. Even though constant values are not altered and the expressions remain semantically equivalent after variable blinding, removing variable names causes a significant drop in performance. This suggests that human-chosen variable names are important and should not be discarded, even when the resulting formula is semantically equivalent.

The graph representation with random edges improves the performance of the 2-hop network, but not for networks with four or more hops. This suggests that the addition of random edges increases the field-of-view into the graph for the networks, but the benefits diminish and ultimately are harmful for networks with more hops.

Subexpression sharing on ASTs results in a directed acyclic graph, so we can experiment with restricting message passing to flow only from children to parents (bottom up), or from parents to children (top down). While allowing information to flow in both directions results in the best performance, Table 2 shows that top down message passing significantly outperforms bottom up. This experiment demonstrates that the context of all the places a subexpression occurs in the larger formula may be more important than the value of the subexpression itself. It's worth noting that this is opposite the bottom up direction of information flow used in TreeRNNs, where information starts at the leaves and then is propagated to the root. However, our neural networks have the benefit of aggregating over all nodes in the graph after message passing is complete, a step not taken in TreeRNNs.

While leaf sharing had the effect of compressing the tree representations by approximately 40%, this representation also incurred a large drop in performance. Most notably, leaf sharing models which used message passing were well below the 0-hop baseline in all cases. This result demonstrates the huge impact the representation has on the GNN performance.

Analyzing only the best-performing representation (subexpression sharing, indicated in bold), we find that models with more hops not only close more proofs, but also find slightly longer proofs on average (1.93 steps for 12-hops vs 1.86 for 0-hops). Another indicator that the quality of the choices made by GNNs increases with with the number of hops is that the percentage of successful tactic applications increased from 33.8% for 0-hops, to 38.6% for 12-hops.

5.3 Implementation Details

In this section, we describe our implementation and hardware setup, including hyper-parameter settings and a more detailed description of the losses we used. These hyper-parameters did not vary between experiments unless otherwise noted, and are presented here to aid reproducibility.

Choosing Negative Examples. For each theorem in the HOList training set, there is only one human proof. We generate positive training examples from each step of the human proof with three values: the goal that is being proved at that step, the tactic applied to it, and a list of theorem parameters passed to the tactic (or a special token if no parameters were

used). To generate positive (goal, premise) pairs, we sample a premise from this list of theorem parameters. In each batch, we sample 16 positive (goal, premise) pairs from our training data.

Because we only have successful human proofs, we select our negative training examples by sampling uniformly from the set of all theorems that were used at least once as positive training examples. Every goal in the batch is then paired with 15 randomly sampled theorems and labeled as negative examples (adding 15x16=240 negative examples to the batch).

Since the combiner network is quite small relative to the GNN embedding networks, we reuse all 256 embedded premises (positive and negative) as negative premise examples for the remaining goals in the batch (adding 15x256=3,840 negative examples to the batch), giving a total batch size of 4,096.

Loss Functions.   In addition to the cross-entropy loss for both tactics and pairwise scores as described in Section 4, we also use the AUCROC loss (Burges et al. 2005; Eban et al. 2017) to directly relate positive and negative premise examples within a batch. The AUCROC loss minimizes the area under the ROC curve and is implemented as follows.

$$AUCROC_b = \sum_i \sum_j loss(logit_i - logit_j)$$

$$loss(l) = \ln(1 + e^{-l})$$

Where i ranges over the positive premises in batch b, and j ranges over the negatives in b. Because our final prediction task is to rank premises for just one given goal, we double the loss for logits that compare positive and negative premises for the same goal.

For the total loss, we take a weighted sum of the cross-entropy loss on the tactic classifier (weight = 1.0), the cross-entropy loss on the pairwise scorer (weight = 0.2), and the AUCROC loss (weight = 4.0).

Optimizers, Learning Rate, Dropout, and Polyak Averaging.   For training, we use an Adam Optimizer (Kingma and Ba 2014) with initial learning rate 0.0001 and decay rate 0.98. Excluding the two GNNs, dropout is added before every dense layer of the network with a "keep probability" of 0.7. For our evaluation checkpoints, we also use moving exponential parameter averaging with rate 0.9999 per step (Polyak 1990; Polyak and Juditsky 1992).

GNN Hyperparameters.   The token features x_t are trainable embedding vectors of size 128. The edge labels $l_E(e)$ are non-trainable binary values $\{0, 1\}$ which indicate if the edge connects a left child or the right child. The graph neural network begins by projecting the node features $x_{l_V(v)}$ and edge features $l_E(e)$ to vectors of size 128 using an MLP with two hidden layers of sizes 256 and 128 with ReLU activations (MLP_V and MLP_E). The resulting embeddings are denoted by h_v^1 and h_e. We then perform t rounds of message passing as per the equations shown in

Section 4.1. MLP_{edge}, $\hat{MLP}_{edge}$ and MLP_{aggr} have an identical configuration with two layers, with hidden sizes 256 and 128 and ReLU activations. The MLPs do not share weights across message passing steps. The final node embedding has size 128. A dropout of 0.5 is applied to all MLPs.

The node embeddings h_v returned by the graph neural network are then aggregated into a single vector that represents the embedding of the entire graph. Using two Conv 1×1 layers, we expand the 128 dimensional node embeddings to 512, and then 1024, with ReLU activations and a dropout rate of 0.5. Then we perform max pooling over all node embeddings to create a single vector of size 1024.

Hardware.   We used eight NVIDIA Tesla V100 GPUs for distributed training, an additional GPU was used purely for evaluation, and we maintained a separate parameter server on a CPU machine.

6   Conclusions and Future Work

We present the first use of graph neural networks to guide higher-order theorem proving. We experiment with several graph representations of higher-order logic expressions and demonstrate that graph neural networks can significantly improve over non-structured representations, especially when the structure of the graph allows for sharing of sub-expressions. We observed that increasing the number of message passing steps resulted in improved accuracy on the training and evaluation sets, and also led to a larger percentage of closed proofs. Using GNNs, we are able to significantly improve previous state-of-the-art results for imitation learning on the HOList theorem set and proof search environment.

In the experiments presented in this paper, we predict tactics and their arguments by looking only at the conclusion of the current sub-goal and ignoring any local assumptions that could be crucial to the proof. This is a serious limitation for our system, and in future work we would like to include the local assumptions list when generating the embedding of the goal. We expect this to be a natural extension for GNNs, because they can easily extend graph representations to include additional expressions.

References

Bansal, K.; Loos, S. M.; Rabe, M. N.; Szegedy, C.; and Wilcox, S. 2019. Holist: An environment for machine learning of higher-order theorem proving. *ICML 2019. International Conference on Machine Learning*.

Battaglia, P. W.; Hamrick, J. B.; Bapst, V.; Sanchez-Gonzalez, A.; Zambaldi, V. F.; Malinowski, M.; Tacchetti, A.; Raposo, D.; Santoro, A.; Faulkner, R.; Gülçehre, Ç.; Song, F.; Ballard, A. J.; Gilmer, J.; Dahl, G. E.; Vaswani, A.; Allen, K.; Nash, C.; Langston, V.; Dyer, C.; Heess, N.; Wierstra, D.; Kohli, P.; Botvinick, M.; Vinyals, O.; Li, Y.; and Pascanu, R. 2018. Relational inductive biases, deep learning, and graph networks. *CoRR* abs/1806.01261.

Bentkamp, A.; Blanchette, J. C.; Cruanes, S.; and Wald-mann, U. 2018. Superposition for lambda-free higher-order logic. In Galmiche, D.; Schulz, S.; and Sebastiani, R., eds., *Automated Reasoning*, 28–46. Cham: Springer International Publishing.

Burges, C.; Shaked, T.; Renshaw, E.; Lazier, A.; Deeds, M.; Hamilton, N.; and Hullender, G. N. 2005. Learning to rank using gradient descent. In *Proceedings of the 22nd International Conference on Machine learning (ICML-05)*, 89–96.

Dong, H.; Mao, J.; Lin, T.; Wang, C.; Li, L.; and Zhou, D. 2019. Neural logic machines. *ICLR 2019. International Conference on Learning Representations*.

Eban, E. E.; Schain, M.; Mackey, A.; Gordon, A.; Saurous, R. A.; and Elidan, G. 2017. Scalable learning of non-decomposable objectives. *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS)*.

Evans, R.; Saxton, D.; Amos, D.; Kohli, P.; and Grefenstette, E. 2018. Can neural networks understand logical entail-ment? *ICLR 2018. International Conference on Learning Representations*.

Gauthier, T.; Kaliszyk, C.; and Urban, J. 2017. Tactictoe: Learning to reason with hol4 tactics. In *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 46, 125–143.

Gilmer, J.; Schoenholz, S. S.; Riley, P. F.; Vinyals, O.; and Dahl, G. E. 2017. Neural message passing for quantum chemistry. *CoRR* abs/1704.01212.

Gonthier, G.; Asperti, A.; Avigad, J.; Bertot, Y.; Cohen, C.; Garillot, F.; Roux, S. L.; Mahboubi, A.; O'Connor, R.; Biha, S. O.; Pasca, I.; Rideau, L.; Solovyev, A.; Tassi, E.; and Théry, L. 2013. A machine-checked proof of the Odd Order Theorem. In *ITP*, 163–179.

Gonthier, G. 2008. Formal proof–the four-color theorem. *Notices of the AMS* 55(11):1382–1393.

Hales, T.; Adams, M.; Bauer, G.; Dang, T. D.; Harrison, J.; Le Truong, H.; Kaliszyk, C.; Magron, V.; McLaughlin, S.; Nguyen, T. T.; et al. 2017. A formal proof of the kepler con-jecture. In *Forum of Mathematics, Pi*, volume 5. Cambridge University Press.

Harrison, J. 1996. HOL Light: A tutorial introduction. In *FMCAD*, 265–269.

Huang, D.; Dhariwal, P.; Song, D.; and Sutskever, I. 2018. Gamepad: A learning environment for theorem proving. *ICLR 2018. International Conference on Learning Representations*.

Kaliszyk, C., and Urban, J. 2014. Learning-assisted automated reasoning with flyspeck. *Journal of Automated Reasoning* 53(2):173–213.

Kingma, D. P., and Ba, J. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Lederman, G.; Rabe, M. N.; and Seshia, S. A. 2018. Learning heuristics for automated reasoning through deep reinforcement learning. *CoRR* abs/1807.08058.

Li, Y.; Tarlow, D.; Brockschmidt, M.; and Zemel, R. 2015.

Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*.

Li, Y.; Yu, R.; Shahabi, C.; and Liu, Y. 2017. Graph convolutional recurrent neural network: Data-driven traffic forecasting. *CoRR* abs/1707.01926.

Loos, S.; Irving, G.; Szegedy, C.; and Kaliszyk, C. 2017. Deep network guided proof search. *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*.

Paliwal, A.; Loos, S. M.; Rabe, M. N.; Bansal, K.; and Szegedy, C. 2019. Graph representations for higher-order logic and theorem proving. *CoRR* abs/1905.10006.

Polyak, B. T., and Juditsky, A. B. 1992. Acceleration of stochastic approximation by averaging. *SIAM Journal on Control and Optimization* 30(4):838–855.

Polyak, B. T. 1990. A new method of stochastic approximation type. *Avtomatika i telemekhanika* 7:98–107.

Raposo, D.; Santoro, A.; Barrett, D. G. T.; Pascanu, R.; Lillicrap, T. P.; and Battaglia, P. W. 2017. Discovering objects and their relations from entangled scene representations. *CoRR* abs/1702.05068.

Sanchez-Stern, A.; Alhessi, Y.; Saul, L. K.; and Lerner, S. 2019. Generating correctness proofs with neural networks. *CoRR* abs/1907.07794.

Santoro, A.; Raposo, D.; Barrett, D. G. T.; Malinowski, M.; Pascanu, R.; Battaglia, P.; and Lillicrap, T. P. 2017. A simple neural network module for relational reasoning. *CoRR* abs/1706.01427.

Scarselli, F.; Gori, M.; Tsoi, A. C.; Hagenbuchner, M.; and Monfardini, G. 2009. The graph neural network model. *IEEE Transactions on Neural Networks* 20(1):61–80.

Selsam, D., and Bjørner, N. 2019. Guiding high-performance sat solvers with unsat-core predictions. *SAT 2019, The 22nd International Conference on Theory and Applications of Satisfiability Testing*.

Wang, M.; Tang, Y.; Wang, J.; and Deng, J. 2017. Premise selection for theorem proving by deep graph embedding. In *Advances in Neural Information Processing Systems*, 2786–2796.

Wu, Z.; Pan, S.; Chen, F.; Long, G.; Zhang, C.; and Yu, P. S. 2019. A comprehensive survey on graph neural networks. *CoRR* abs/1901.00596.

Xu, K.; Hu, W.; Leskovec, J.; and Jegelka, S. 2019. How powerful are graph neural networks? *ICLR 2019. International Conference on Learning Representations*.

Yang, K., and Deng, J. 2019. Learning to prove theorems via interacting with proof assistants. *CoRR* abs/1905.09381.